0.项目介绍

这个项目是基于B/S架构，在Linux环境下搭建服务器端，服务器端可以对浏览器客户端发来的http请求进行解析，根据浏览器客户端的具体要求返回响应。服务器端利用epoll IO复用技术实现Reactor模型，其中主线程负责对多个客户端进行监听，并将客户端的任务添加到线程池的任务队列中，线程池中的子线程到任务队列中取任务去执行，为了减少系统资源浪费，还实现了可自动扩容的服务器缓冲区和小根堆定时器。服务器搭建完成后使用webbench测试工具对服务器的性能进行测试，在4核4gb的乌班图环境下，并发量可以达到8000。

1.线程池

1.1 线程池简介

线程池主要是围绕生产者消费者模型来创建的，生产者是主线程，它向任务队列中添加数据(就是要执行的任务)，消费者是线程池中子线程，从任务队列中取数据(从任务队列中取任务去执行)。任务队列是多线程的共享数据，所以这里要考虑数据安全问题，我是使用互斥锁来保证数据安全问题，(注意互斥锁保护的是整个任务队列，而不是单个任务，加锁函数pthread_mutex_lock*[m'ju:teks]*)，同时也使用了条件变量，因为子线程代码逻辑是写在一个while死循环中，它会不停获得锁，然后访问任务队列取数据，执行任务，当任务队列为空的时候，子线程就不需要再到任务队列中取任务了，所以使用条件变量阻塞子线程，当任务队列中再有任务时再唤醒某个阻塞的子线程去竞争锁，然后到队列中取任务执行。

一开始任务队列就是空的，所有子线程都处于阻塞状态，当主线程向任务队列中添加一个任务以后唤醒某一个子线程，该子线程获得互斥锁，然后去任务队列中取任务，从任务队列取出任务后再释放这个锁，之后就可以去执行这个任务的回调函数，任务执行完成后，因为每个子线程是死循环的方式轮询任务队列，所以之后要再次尝试获得锁去任务队列中取任务。

1.1 线程池创建过程

自己创建一个线程池类，在线程池类的构造函数中创建出固定数目的子线程，然后每个子线程具体执行都是放在一个while死循环里面来做的，在while死循环中判断工作队列是否为空，如果不为空那么子线程会去竞争互斥锁资源，只有获得这个锁资源的子线程才能访问共享队列去取任务，然后执行任务的回调函数，如果工作队列为空那么子线程就一直阻塞在线程池条件变量上。在这个类的析构函数中会唤醒所有子线程，并把表示线程池是否关闭的标志位设置为1，然后子线程在执行到标志位为1的逻辑代码时会break跳出死循环，执行detach()线程分离，系统回收子线程资源。

什么是detach线程分离?

线程分离状态：指定该状态，线程主动与主控线程断开关系。线程结束后，其退出状态不由其他线程获取，而直接自己自动释放。**网络、多线服务器常用。**

简单来讲，线程分离就是当线程被设置为分离状态后，线程结束时，它的资源会被系统自动的回收，而不再需要在其它线程中对其进行 pthread_join() 操作。

进程若有这个机制，则将不会产生僵尸进程。僵尸进程的产生主要由于进程死后，大部分资源被释放，一点残留资源仍存于系统中，导致内核认为该进程仍存在。

1.2 为什么创建线程池

因为创建线程和销毁线程都需要调用系统内核的api来完成，这会占用CPU时间，当线程数很多时这个时间很可能比处理业务逻辑的时间还长，如果把线程的创建、销毁以及线程处理业务逻辑所耗费的系统资源都算上，可能就会导致系统资源不足。所以创建一个线程池，在服务器启动之初，就在里面创建好一些线程，当服务器处理完一个客户端连接，再把这个线程资源放回到池中。我们在线程池中创建的线程也不会很多，这些线程只会创建一次，避免了频繁创建和销毁线程带来的开销，通过这样一种空间换时间的方法，虽然浪费服务器少量的硬件资源，但是对整个系统的执行效率有很大提升。

使用生产者-消费者模型的好处是解耦：如果没有共享队列，生产者生产一个数据，如果线程池中没有空闲线程，则生产者需要阻塞等待有空闲的线程以后，再把数据交给工作线程，而有了队列以后，生产者只需要将数据放到队列里面，然后去做其他的事情即可。

1.3 线程池内线程数量

线程池中的线程数量最直接的限制因素是CPU核数量，如果是CPU密集型的任务，一般线程数量设置为和CPU核数相同就行，因为**CPU密集型任务主要是进行一些运算操作**，这时候多线程会根据情况分布在不同的核心上计算，达到充分利用CPU的目的，如果这时候，线程数量过多，可能会使CPU过多的进行线程切换，浪费时间和资源。而我们这里线程池中工作线程除了执行业务逻辑外，还需要进行IO操作读写数据，属于**IO密集型任务**，线程数量设置的要多一些，一般IO的处理比较慢，多于核数的线程会为CPU争取更多的任务，不至于在线程处理IO的过程造成CPU空闲导致资源浪费，同时如果有线程被阻塞，也可以使其让出CPU资源给其他线程使用。

(读写数据时，未满足条件而阻塞等待的IO过程，会让CPU空闲)

1.4 动态增加线程

可以先通过一个测试工具，测量出服务器处理每个客户端请求的正常时间。然后将这个时间作为一个阈值，再通过Linux下的curl命令监控到当前http请求的耗时，如果这个耗时大于之前设置的阈值，可以说明主线程交给工作线程的任务，工作线程没有及时处理，说明线程池满负荷运转，就再去创建更多新的线程。

1.5 线程同步(锁)

线程同步(多个线程之间协同数据):当一个线程对内存中的共享数据（临界资源）进行操作的时候，其他线程不可以对这个数据进行操作，直到该线程完成后才可以，主要是防止多线程对共享资源的竞争造成数据错乱。

1.5.1 线程同步方式

线程同步的方式：信号量、条件变量、读写锁、互斥锁、自旋锁

信号量：信号量可以看作是一个计数器，当有线程访问资源时信号量加1，访问完成时信号量减1(这个修改是原子操作)，它允许同一时刻有多个线程来访问共享资源，但是会控制访问资源的最大线程数，当信号量为0时，访问资源的线程被阻塞，直到信号量大于0。

条件变量：条件变量是线程间的一种通知机制，一个条件变量可以阻塞多个线程，这些线程会组成一个等待队列，直到条件成立后，向等待队列中的一个或所有线程发送“条件成立”的信号，解除它们的“被阻塞”状态。

锁：加锁的目的就是保证多线程条件下，共享资源在任意时间里，都只有一个线程可以访问，解决多个线程因资源竞争最终导致数据错乱的问题。

互斥锁：互斥锁是一种独占锁，当前线程加锁成功，其他线程就会加锁失败被阻塞(内核设置为睡眠状态)，直到锁被当前线程释放了，内核会在合适时候唤醒阻塞线程。//多线程条件下防止共享资源被并发访问，造成数据错乱，同一时间只有一个线程可以访问共享资源，在访问前对互斥量加锁，在访问完成后对互斥量解锁，这样就保护了共享资源。

读写锁：读写锁是一把锁，它有读和写两个状态，当“写锁”没有被线程持有的时候，多个线程可以共同持有”读锁“，读取共享资源。一旦“写锁”被线程持有，那些想要获取“读锁”的线程和想要获取“写锁”的线程都会被阻塞，任意时刻只允许一个线程持有写锁，在读多写少的时候应该选择读写锁。 读共享，写独占。 写锁优先级高。

自旋锁：如果当前线程想要使用自旋锁，但发现锁资源被其他线程持有，当前调用线程会一直忙等待，不停地循环判断锁资源是否被释放，直到锁被释放让当前线程获得为止。在确定加锁部分的代码执行时间很短时，使用自旋锁比较合适。

互斥锁和读写锁的区别：

对于互斥锁，无论是读线程还是写线程，任意时刻都只能有一个线程获得锁资源，加锁失败的线程会被阻塞，直到锁资源被释放。读写锁是比互斥锁粒度更小的锁，它是一把锁具有读锁和写锁两种状态，它允许多个线程同时持有读锁，这样可以提高共享资源的访问效率，增加读操作的并发性，但是在写锁状态下只允许有一个线程可以持有，所以读写锁适合于读多写少的场景。

互斥锁和自旋锁的区别：

(1)互斥锁加锁失败以后，线程会释放cpu资源，让给其他线程，自己被阻塞。自旋锁加锁失败以后，线程会忙等待，不停的去检查锁资源是否被释放，直到锁被释放，该线程获得锁资源。

(2)互斥锁的开销比自旋锁大，如果对互斥锁加锁失败，需要进行线程切换来应对，操作系统会把加锁失败的线程由运行态设置为睡眠态，再把CPU切换给其他线程运行，等到锁被释放了，操作系统把处于睡眠态的阻塞线程变为就绪态，再把CPU切换给该线程使用。而自旋锁一般是通过CPU提供的CAS函数在用户态完成加锁和解锁操作的，开销比较小。

1.5.2 线程通信时是否可以将锁换成布尔变量实现同步

不可以，因为这个布尔值也是共享的，多个线程可以同时去修改这个布尔值，当一个线程获得这个布尔值以后，把它置为1，表示这个线程要访问共享资源，然后进入任务队列取数据，这个时候其他子线程过来可以直接对布尔值进行修改，然后也进入到任务队列中，这个布尔值并不能够起到阻塞线程，实现同步的作用。

1.6 CAS

CAS(compare and swap)

CAS(对比并交换）原理：CAS是一个原子指令，一般是用在多线程环境中实现同步。 它包括三个操作数：内存位置(V)、预期原值(A)、新值(B)。如果内存位置的值与预期原值相匹配(说明没被其他线程修改，现在可改)，那么处理器会自动将该位置值更新为新值，否则的话，不做任何操作。这个过程是作为单个原子操作完成的。原子性保证了这个新值是最新的，并且这个新值在被读取到操作完成过程中不会被其它线程修改。

CAS有效地说明了“我认为位置V应该包含值A；如果包含该值，则将B放到这个位置；否则，不要更改该位置，只告诉我这个位置现在的值即可。

什么是原子操作/指令?

简单来说，`原子操作（atomic）`就是不可分割的操作，在计算机中，就是指不会因为线程调度被打断的操作。

比如，简单的赋值是一个原子操作：

> m = 6; // 这是个原子操作

假如 m 原先的值为 0，那么对于这个操作，要么执行成功 m 变成了 6，要么是没执行 m 还是 0，而不会出现诸如 m=3 这种中间态 —— 即使是在并发的线程中。

而，声明并赋值就不是一个原子操作：

> int n = 6; // 这不是一个原子操作

对于这个语句，至少有两个操作：
①声明一个变量 n
②给 n 赋值为 6
—— 这样就会有一个中间状态：变量 n 已经被声明了但是还没有被赋值的状态。
—— 这样，在多线程中，由于线程执行顺序的不确定性，如果两个线程都使用 m，就可能会导致不稳定的结果出现。

1.7 死锁原因

多个进程在执行过程中，因为争夺资源出现了进程相互之间无限期阻塞等待的状态就是死锁。

死锁发生的情况：

(1) 加锁之后忘记解锁，下次再来访问时也被阻塞住，造成死锁

(2) 线程A、B，分别加锁了a、b两种资源，此时A尝试去访问b，B尝试去访问a，会造成AB同时被阻塞，造成死锁（循环等待）

(3) 对某一资源重复加锁，第二次加锁时会阻塞住，造成死锁

产生死锁的前提条件:

(1)互斥条件，每个资源一次只能一个进程使用，其他进程想使用需等待该进程释放。

(2)占用等待条件（请求与保持条件），每个进程都会占用一个资源，同时每个进程还在等待另外一个被其他进程所占用的资源。

(3)非抢占条件（不剥夺条件），进程被分配的资源，只能自己释放，不能被强制抢占

(4)循环等待条件，多个进程之间形成了一个环形的资源等待的关系。

1.8 死锁解决

解决：

(1)资源一次性全部分配，让每个进程都得到他想要的资源

(2)当发生死锁时，终止某些进程，释放它占有的资源

(3)资源可以按照一种顺序逐个分配给请求资源的进程

1.9 死锁检测

检测：可以给每个进程和每个资源都制定唯一编号，创建一张资源分配表，记录各进程与占用资源之间的关系。再创建一张进程等待表，记录各进程与要申请资源之间的关系，如果出现环路等待情况，就是发生了死锁。

可以gdb调试死锁：先用ps命令查看进程id，然后attach进程，(调试正在运行的进程， gdb attach pid)，再info threads看各个线程的状态（thread apply all bt：查看所有线程的堆栈信息，print mutex_name 查看锁的信息）看是哪些线程在等待锁，再进入线程，使用bt命令，看线程堆栈情况，可以查看具体哪些函数有问题再去查看代码

1.10 八个子线程，如何实现高并发

我们子线程处理业务逻辑的代码是写在while循环中的，线程池中的每一个子线程处理完当前任务就去处理下一个，没有任务就一直阻塞在那里（条件变量上）等待。同一时刻8个线程都在处理请求任务，处理完一个任务后接着处理下一个，直到请求队列为空表示任务全部处理完成，通过这种方式实现的服务器高并发。

线程池类的析构函数中，设置了一个关闭线程池的标志位，如果标志位值为true，子线程会从while循环中跳出，调用detach将线程进行分离，系统回收子线程资源。

2.小根堆定时器(堆原理)

2.1 为什么用小根堆实现定时器

增加定时器功能主要是为了关闭超时的非活跃连接，定时器的实现一般有三种方式，升序双链表、时间轮(环形数组+单链表)、小根堆。从定时器执行效率方面考虑，选择的数据结构最好是有序的，能够快速找到当前最小的超时时间，然后执行超时任务，小根堆的根节点存放的就是最小的超时时间，所以O(1)复杂度就可以做到，同时我们可能会向定时器容器中频繁的插入和删除超时任务，小根堆插入和删除超时任务的时间复杂度都是Olog(n)，双向升序链表增加超时任务是O(n)、删除和执行都是O(1)，时间轮的话增加和删除超时任务都是O(1)、执行超时任务是O(n)，所以综合来看小根堆的效率比较高。从定时精度方面来对比，升序双向链表和时间轮都是固定的频率去检查是否有定时器超时，然后去执行超时任务(回调函数)，而小根堆的话，是以当前根节点中最小的超时时间为单位去检查是否有定时器超时，这个时候定时时间最小的定时器一定是超时的，然后去执行定时器任务，执行完再取出下一个最小的超时时间作为下次检测的时间间隔，如此反复就实现了相对精准的定时。

kkby:
如果是阻塞timeMS那说明一直没有新的事件发生，下面的eventCount就是0，后面的循环也不执行，时间几乎就等于timeMS

kkby:
如果有事件发生，那调用tick的时间取决于每次epollwait等待的时间和后续程序运行的时间，没有一个固定值

(执行效率指的是增加定时任务、删除定时任务以及执行定时器任务(检查最近要发生的定时任务))

2.2堆的原理

堆从内存视角来看，它是一个连续存放的数组，而从逻辑视角来看它是一颗顺序存储的完全二叉树，堆又分为大根堆和小根堆：大根堆是指在完全二叉树当中，所有子树的根节点值都要大于等于它的左右子树节点的值。小根堆是指完全二叉树当中，所有子树的根节点值都要小于等于它的左右子树节点的值。

2.3小根堆具体实现

小根堆定时器实现，首先是通过add函数来建堆，当一个客户端有数据到达时，我们把它的定时时间和文件描述符传到add函数中，然后判断这个文件描述符是否存在于小根堆定时器中，如果是新的节点堆尾插入，插入后不断和父节点比大小，调整节点位置，满足小根堆的特性。(父亲索引只要大于等于0,就去判断，我父结点的值是否比我当前节点的值要小,如果要小的话就break，不用更换,否则交换两个节点的值)，如果是已有节点，对节点的定时时间变量重新赋值，赋值完再调整堆。

堆删除节点（del函数）：首先判断待删除结点是否为堆尾节点，如果是直接pop移除该节点，如果不是则将要删除的节点和堆尾节点进行交换，要删除节点位于堆尾，重新调整堆中元素顺序，调整顺序时，不包含那个待删除的位于堆尾的节点，然后直接pop堆尾节点完成删除操作。

2.4 如何触发定时器

只要服务器不关闭，主线程就要一直调用epoll_wait()来监听是否有数据到达，定时器设定的时间是epoll_wait()的阻塞等待时间（epoll_wait()的阻塞等待时间就是定时器的根节点即最小的那个时间），如果被epoll_wait检测的文件描述符，超过这个时间了还没有数据到达，程序就向下继续执行，执行结束后，当再次回到while循环的开始处时就会调用定时器的触发函数gettick()。

在主线程里面会调用一个gettick()函数，这个函数会判断当前小根堆中堆顶的定时时间是否已经超时，如果超时，会去执行超时节点的回调函数，会将这个文件描述符从epoll_wait的检测集当中删除，并关闭这个客户端文件描述符，再把这个节点从小根堆定时器里面pop()出去，调整小根堆结构，再次获取定时时间最小的节点，把他的定时时间重新赋值给epoll_wait()。

2.5 怎么定时的/怎么判断超时的

我的定时时间=当前北京时间+设置的超时时间，

利用相对值的方式来判断是否超时，设置好定时时间，经过一段时间后去检查，用定时时间减当前北京时间，看这个差是否小于0，如果小于0说明超时了，进行超时处理，将文件描述符从epoll的事件集中摘除，然后关闭这个文件描述符。

2.6 如何知道客户端断开连接

检测用于通信的文件描述符对应的epoll事件是**EPOLLRDHUP**时，就说明客户端关闭了连接

3.http请求与响应

函数调用流程：

主线程while循环监听——Deallisten（accept）——AddClient

主线程while循环监听——DealRead（threadpool_->AddTask）——OnRead（1.read 2.OnProcess）——1.HttpConn::read()分散读——2.WebServer::OnProcess解析并生成响应，生成完响应修改fd为EPOLLPUT

主线程while循环监听——DealWrite（threadpool_->AddTask）——OnWirte（write分散写）——HttpConn::write()分散写

1.http连接的处理

服务器启动以后，会先创建好线程池。当浏览器客户端发出http连接请求，每个客户端连接进来以后都会封装成HttpConn类型的对象保存到map容器中，key存放的是客户端文件描述符，value存放的是客户端的信息(HttpConn对象，一个存放协议、ip地址、端口号的结构体变量)，然后就将这个客户端连接注册到epoll的内核事件表中，之后epoll_wait会监听这个客户端文件描述符是否有事件发生。

具体：
在主线程Webserver.Start()函数的while循环中通过epoll_wait函数监听到有新客户端连接时，调用DealListen函数处理监听，在DealListen函数中调用accept [əkˈsept]函数真正接收客户端连接，因为我们是ET模式，所以这个调用accept函数实际上是在一个while循环中进行的，这样就保证一次把连接的所有的客户端都处理了。使用accept接收客户端后再调用AddClient函数，在这个函数中主要进行三个操作，分别是创建新的客户端连接对象并将其加到Webserver服务器类的map统计数组上、为新连接的客户端创建定时任务并挂载到小根堆定时器上、将新连接的客户端添加到epoll对象中进行监听并设置文件描述符为非阻塞，至此，处理新客户端连接的流程就全部完成了。

2. 读请求报文的过程?

如果文件描述符上有可读事件发生，主线程会将任务添加到工作队列中，睡眠在工作队列上的子线程会被唤醒，子线程执行任务，将客户端的请求报文数据读入到客户端对象对应的缓冲区中(每个连接被封装成了http类对象，它的文件描述符和信息放到了map里)，数据读好以后，通过有限状态机搭配正则表达式进行解析。

具体：

在主线程Webserver.Start()函数的while循环中通过epoll_wait函数监听到有客户端fd有数据到达时（可读事件发生），调用DealRead函数处理监听，在DealListen函数中首先延长小根堆定时器中该客户端的定时时间（实际上就是更新），然后通过Addtask函数将该客户端以及OnRead这个函数添加到任务队列中，唤醒一个线程池中的线程从任务队列中取这个任务，然后去执行这个OnRead函数，在OnRead函数中主要进行两个操作，一是调用read函数将客户端发来的数据存放在我们定义的可扩容缓冲区中（因为ET模式，所以是循环读直到读完），二是对接收到的数据通过有限状态机搭配正则表达式进行解析。

3.具体是怎么读的？（自动扩容缓冲区是什么？怎么读到里面去的？）

如果文件描述符上有可读事件发生，主线程会将任务添加到工作队列中，睡眠在工作队列上的子线程会被唤醒，子线程执行任务，将客户端的请求报文数据读入到客户端对象对应的缓冲区中

具体：

我定义了自动扩容缓冲区的类，连接进来的客户端都被封装成了一个HttpConn客户端连接类的对象，在这个HttpConn客户端连接类中我们定义了两个自动扩容缓冲区类的对象，分别表示读缓冲区和写缓冲区。当进行读操作时，调用读缓冲区对象的read函数，在这

个函数中是通过分散读的方式来进行读的，创建一个iovc的数组，这个数组的第一个元素和第二个元素表示的就是分散读的两个目标位置，设置第一个元素为自动扩容缓冲区的vector类型的存储数组（也就是实际的缓冲区），第二个元素为一个65536字节大小的buff临时数组（设置这么大的原因是保证能够把所有的数据都读出来），然后调用readv函数分散读，当客户端传来的数据大小小于vector存储数组剩余空间时就直接读到vector存储数组中去了，大于vector存储数组剩余空间的话，多出来的数据会被读取到临时数组中，然后根据临时数组中数据的大小对vector存储数组进行扩容，最后再把临时数组中的数据拷贝到vector存储数组中，实现自动扩容，这样就完成了数据的读取过程。

4.如何解析请求报文（有限状态机搭配正则表达式是如何解析报文的？）

如果文件描述符上有可读事件发生，主线程会将任务添加到工作队列中，睡眠在工作队列上的子线程会被唤醒，子线程执行任务，将客户端的请求报文数据读入到客户端对象对应的缓冲区中(每个连接被封装成了http类对象，它的文件描述符和信息放到了map里)，数据读好以后，

在HttpConn客户端连接类中有一个HttpRequest Http解析类的成员变量，解析报文时，会调用这个HttpRequest成员变量的解析函数，依次解析客户端读缓冲区中的数据。具体是使用有限状态机搭配正则表达式的方式来做http报文的解析，通过有限状态机标志位的改变(switch case)来执行报文不同部分的解析，即请求首行、请求头、请求体的解析。利用regex_match /ri 'geks meh/ 函数根据定义的正则规则去匹配。[]:表示匹配中括号中的字符，[^]表示匹配除了中括号中^后面字符以外的字符，*表示匹配前面的子表达式零次或多次，?表示匹配前面的子表达式零次或一次。

```
    //解析请求头：GET / HTTP/1.1
    regex patten("^([^ ]*) ([^ ]*) HTTP/([^
]*)$");//利用正则来解析，就是定义一个规则(正则表达式)去匹配字
符串
    smatch subMatch;
//bool regex_match():在整个字符串中匹配到符合整个表达式的整
个字符串时返回true，也就是匹配的是整个字符串。
    if(regex_match(line, subMatch, patten)) {//line
是我们要去匹配的字符串，我们匹配的结果数据会保存到subMatch
中，patten是定义的正则表达式(按照怎样的规则匹配)
        method_ = subMatch[1];//请求首行中的请求方法
        path_ = subMatch[2];//请求首行中的请求路径
        version_ = subMatch[3];//请求首行中的协议版本
        state_ = HEADERS;//解析完状态首行后状态发生改变
        return true;
    }

//解析请求头:regex patten("^([^:]*): ?(.*)$");
```

5.如何知道数据已经读完

因为项目中EPOLL选择的是ET工作模式，所以会放在while循环中读，一直到把内核缓冲区的数据全部读取完才结束，读取完后会返回读到的数据数量继续往下执行，或者当前没数据可读会返回EAGAIN，然后继续往下执行，如果返回值是-1或者errno不是EAGAIN的话说明读取出错了，会关闭客户端文件描述符，把这个文件描述符从内核监听事件集中删除。（这一步是在主线程监听到读事件，将读任务挂到任务队列，子线程取任务执行OnRead，OnRead调用HttpConn的read函数，在这个read函数中完成的）

6.如何判断请求体结束?

http的解析也是放到while循环中去做的，判断条件是只要自定义的缓冲区中有数据没处理或是有限状态机的标志位不为Finish就继续解析。

另外可以让服务器在收到数据之后，将实际收到的数据量与报文头部中的Content-Length字段对应的值作比较，来判断请求体的数据是否全部收到了。

7.收到不完整请求报文，怎么办

目前没有考虑接受不全的问题，因为服务器只是完成基础的get和post请求，不会有过多的数据量发送过去，TCP缓冲区可以足够接收，一次性的把请求数据全部接受。但对于这种情况，如果未来想要实现的话，我觉得可以这样，就是我先把这次读到自定义用户缓冲区的请求报文进行解析，如果发现报文不完整，可以把缓冲区的读位置索引还原回去(read指针)，然后收到下一次数据再继续解析。

目前只是简单的解析了表单中的信息，暂时不支持文件上传。

8. 生成响应报文的过程?

在HttpConn::process()函数中调用HttpConn::prase()函数解析请求成功后，会调用HttpResponse::Init()初始化HttpResponse类类型的响应对象，主要是根据解析出来的信息初始化响应对象的资源路径、资源目录等，然后调用HttpResponse::MakeResponse()函数生成响应，在这个函数里会根据请求的资源路径和目录是否存在（通过stat函数获取文件信息，如果返回值小于0说明不存在），依次生成相应的响应状态行、响应头、响应体等信息（注意：响应行和响应头是放在HttpConn::Writebuff写缓冲区的，而响应体即文件是通过mmap映射到进程的一块内存上的，这也是返回响应时需要分散写的原因）。

9.返回响应的过程?

在WebServer::OnRead()函数中解析请求、生成响应完成后，会将该客户端文件描述符添加到epoll对象的写监听集合中，监听写事件，当检测到客户端可写时，调用DealWrite函数处理监听，在DealWrite函数中首先延长小根堆定时器中该客户端的定时时间（实际上就是更新），然后通过Addtask函数将该客户端以及OnWrite这个函数添加到任务队列中，唤醒一个线程池中的线程从任务队列中取这个任务，然后去执行这个OnWrite函数，在OnWrite函数中将生成的响应通过分散写的方式写回客户端。

10.为什么分散写？分散写的具体过程?

我们在生成响应的时候，响应首行、响应头是放在HttpConn客户端连接对象的写缓冲区里的，而为了提高访问速度，响应体也就是文件的具体内容是通过mmap映射到一块内存中的，也就是说，响应的内容是分散在两块不同的内存中的，所以需要分散写。

创建一个iovc的数组，这个数组的第一个元素和第二个元素表示的就是分散写的两个源位置，设置第一个元素为HttpConn连接对象的写缓冲区（内部保存的是响应首行和响应头），第二个元素为mmap映射的文件指针，然后调用writev函数分散写。

11.为什么采用mmap内存映射？什么是mmap内存映射?

采用mmap内存映射的原因是为了提高对文件的访问速度，如果不采用mmap内存映射，那么对文件的读取只能用read函数，而read函数是一个系统调用，使用其会进行用户态和内核态的切换，访问速度较慢。

mmap是一种内存映射文件的方法，即将一个文件或者其它对象映射到进程的地址空间，实现文件磁盘地址和进程虚拟地址空间中一段虚拟地址的一一对映关系。实现这样的映射关系后，进程就可以采用指针的方式读写操作这一段内存，而系统会自动回写脏页面到对应的文件磁盘上，即完成了对文件的操作而不必再调用read,write等系统调用函数。相反，内核空间对这段区域的修改也直接反映用户空间，从而可以实现不同进程间的文件共享。

12. 内核态/用户态?

内核态：也叫**内核空间**，是内核进程/线程所在的区域。**主要负责运行系统、硬件交互。**

用户态：也叫**用户空间**，是用户进程/线程所在的区域。**主要用于执行用户程序。**

内核态：内核空间存放的是操作系统内核代码和数据，它是被所有程序共享的，在程序中修改内核空间中的数据不仅会影响操作系统本身的稳定性，还影响其他程序，所以操作系统禁止用户程序直接访问内核空间。

用户态：用户空间保存的是应用程序的代码和数据，是程序私有的，其他程序一般无法访问。

（问内核态与用户态的区别时只回答上面就行）

1.内核和用户程序共用地址空间

让内核拥有完全独立的地址空间，就是让内核处于一个独立的进程中，这样每次进行系统调用都需要切换进程。切换进程的消耗是巨大的，不仅需要寄存器进栈出栈，还会使CPU中的数据缓存失效、MMU中的页表缓存失效，这将导致内存的访问在一段时间内相当低效。

而让内核和用户程序共享地址空间，发生系统调用时进行的是模式切换，模式切换仅仅需要寄存器进栈出栈，不会导致缓存失效；现代CPU也都提供了快速进出内核模式的指令，与进程切换比起来，效率大大提高了。

2.何时会从用户切换到内核

1.系统调用：用户进程通过系统调用申请操作系统提供的服务程序来完成工作。

2.发生异常：当CPU在执行运行在用户态的程序时，发生某些不可知的异常，就会触发由当前运行进程切换到处理这个异常的内核相关程序，也就到了内核态，比如**缺页异常**。

3.外围设备的中断

·当外围设备完成用户请求的操作之后，会向CPU发出相应的中断信号，这时CPU会暂停下一条要执行的指令，转去执行中断信号的处理程序，如果先执行的指令是用户态下的程序，那么这个转换的过程自然也就发生了从用户态到内核态的切换。

4.自动扩容缓冲区

3.1扩容缓冲区基本原理

这个缓冲区指的是服务器的用户区缓冲区，因为我们要对浏览器客户端发来的请求数据进行解析，从客户端传过来的数据，最先达到服务器的socket读缓冲区当中，所以要把数据读到用户区才能去做报文解析，这就有一个问题，**一个客户端每次发来的请求数据的大小可能会有变化，这个用户区的缓冲区开辟多大空间合适呢？**所以我们把这个缓冲区的大小变成一种动态的方式，可以自动扩容。

首先定义一个char型的vector作为用户区自定义的缓冲区，它的大小默认是1024字节，使用双指针的方式来标记这个缓冲区的可读起始位置(下次读的时候从这个位置开始读)和可写起始位置(下次写的时候从这个位置开始写)，这个缓冲区中可写字节数的大小就是缓冲区中写标志位（写指针）之后的空间，缓冲区可读字节数大小就是读标志位（读指针）与写标志位（写指针）之间的空间。然后还要定义一个char类型的临时数组，它的大小为65536。

这个临时数组用来帮助1024vector实现扩容的，当我们从内核区的socket缓冲区读数据，读到用户缓冲区，这时分为两种情况，如果读到的数据大小比1024vector的可写数据要小的话，直接读到1024vector中即可，如果要是比它大的话，说明光靠1024vector装不下读到的数据，那就得读到临时数组里(先把1024装满，再把装不下的放到临时数组里)，然后去实现1024vector的扩容操作，扩容也分两种情况，第一种情况是这个1024vector缓冲区中的数据被解析掉了一部分，那么这部分空间也可以被用来存放新数据，如果前面这部分解析完的空间和后面可写空间相加比临时数组中的数据的字节数大的话，就把当前缓冲区中读到的数据，拷贝到最前面，相当于移动到最前面索引为0的位置上，把前面解析过的数据覆盖掉，这样前面解析完的字节空间就增加到了缓冲区的后面，缓冲区可写字节数就增加了，这时再将65536临时数组里面的内容拷贝到1024缓冲区后面空余的位置上。另外一种情况就是1024 vector前面解析完的字节空间与后面缓冲区可写字节空间加在一起的大小还是比临时数组中存放的数据小，那就调用vector的resize进行扩容,resize后的大小就是原来vector所占字节数加上临时数组中的字节数。扩容完之后，将临时数组里面的内容拷贝到vector缓冲区新增加的空间里。随着客户端发来的请求报文数据的变化，不断的增长缓冲区的大小，通信次数达到一定程度以后，最终增长的缓冲区的大小就会固定在一个比较合适的值。

3.2实现扩容缓冲区原因

(为啥不直接分配一块大内存作为缓冲区、自扩容缓冲区多次分配内存不会产生很多内存碎片吗)

回答:1、可以考虑直接分配一块大空间作为缓冲区，但是申请多大合适呢?

回答2、对缓冲区大小调整的操作并不会执行太多次，因为一个连接通信几次后，他每次的数据量差别不会太大。

5.Raii机制

Raii翻译过来是资源获取即初始化，它是C++管理资源、避免内存泄漏的一种方法，C++中创建对象时会自动调用构造函数，对象超出作用域时会自动调用析构函数，RAII就是使用一个对象，在它构造的时候获取对应的资源，在对象生命周期内控制对资源的访问，在对象析构的时候释放构造时获取的资源。

6.数据库连接池

调用：main:WebServer MyWebServer() ->SqlConnPool::Init()

在项目中我们有一个数据库连接池类，通过get和free成员函数可以从MYSQL队列(池）中获取数据库连接或者放回数据库连接，这个队列也是多个线程的共享数据，所以使用了互斥锁来对其保护，只有获得互斥锁的线程才能访问队列(池)获取数据库连接，同时也使用了信号量，如果队列中没有数据库连接，信号量为0，会阻塞线程。我们的数据库连接池类采用单例模型来保证只有一个数据库连接池对象，同时使用C++11静态局部变量的特性来保证单例模式的线程安全问题（C11规定，当一个线程对一个静态局部变量进行初始化时，如果其他并发的线程也进入到函数想对这个局部静态变量初始化，其他线程会被阻塞，直到之前的那个线程完成对静态局部变量的初始化），定义一个RAII类，将数据库连接池对象(其实就是

SqlConnPool::Instance()函数的返回值）作为RAII类有参构造的形参，在其构造函数中调用get函数从数据库连接池中（队列）获取一个数据库连接资源，在其析构函数中调用free释放数据库连接资源，这样做对数据库连接资源的管理更加方便(在创建RAII类的匿名对象时会自动调用有参构造获取数据库连接，对象超出作用域时，这个对象会自动调用析构函数，释放连接。)

数据库连接池也是围绕生产者-消费者模型来创建的，它的生产者和消费者指的都是子线程，一开始创建出10个数据库连接，之后子线程来队列中获取数据库连接就是消费者，把连接放回队列就是生产者，当信号量为0会阻塞当前线程，直到其他线程放回数据库连接到队列中，被阻塞线程才解除阻塞状态。

7.压力测试

6.1 正常步骤

使用webbench工具来进行压力测试，在命令行里面运行压测工具代码，指定访问的客户端数目和访问时间，Webbench基本原理是fork出多个子进程，每个子进程循环做Web访问测试。子进程把访问结果通过管道告诉父进程，父进程统计最终结果，通过结果可以知道服务器每分钟输出的页面数和每秒传输的字节数(10w或百万级别)，以及成功建立的连接数量和失败的连接数量。

测试示例：

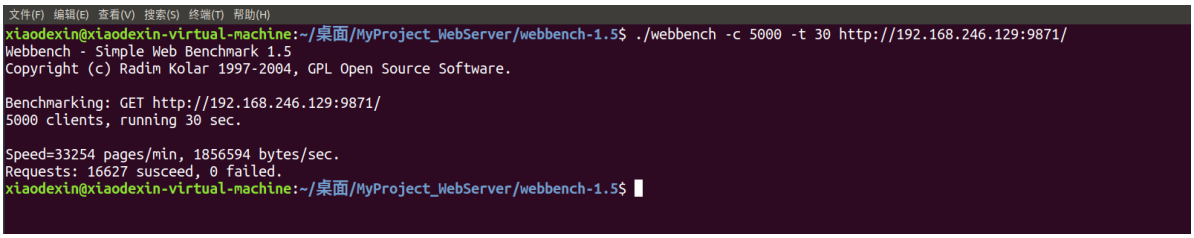
./webbench -c 5000 -t 30 http://192.168.246.129:9871/（服务器ip地址端口号，访问资源的路径)

参数:指定客户端数量和访问时间

-c表示客户端数(fork出的子进程数量),并发量，同时有这么多客户端向服务器进行访问

-t 表示时间(30表示30s)

正确的结果：



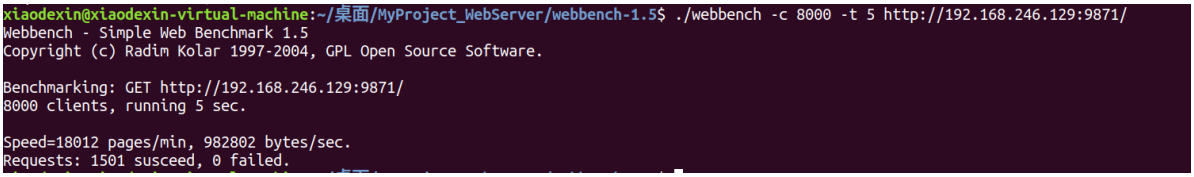
指标:speed

pages/min表示每分钟访问的页面数，bytes/sec表示每秒传输的字节数，

Requests:成功处理的请求数，failed：失败的请求的数。

每分钟访问多少页面，每秒访问的字节数(数据量)

测试：



8k个客户端5s内成功请求次数1501次，访问速度是每分钟访问超过1.8w个页面，每秒访问字节数近10W

6.2 发生错误

在编写代码过程中，一般是开一两浏览器窗口去访问服务器，这时候没出现什么问题。

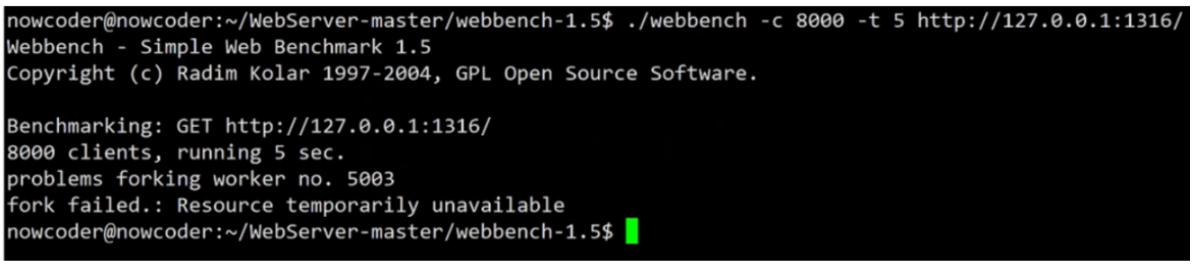
最后压力测试阶段，压力测试结果显示的是出现了问题（有一个报错信息fork faild），webbench的原理就是父进程同时创建很多子进程来向服务器发起连接，然后大量客户端同时发起连接导致服务器不能正常工作,出现这个问题我最先想到的是服务器并发处理大量请求时候，如果TCP全连接队列过小，就容易溢出，TCP全连接队列溢出，后续的请求就会丢失，这样就会出现服务端请求数量上不去的现象。listen()系统调用中的backlog参数表示的就是同时和服务器建立连接的客户端的上限数，所以想着把它改大应该可以，但是

改大以后依然不行，后来了解在Linux下用tcpdump这个工具，用它来获取数据包，再使用Wireshark来分析这个数据包中的数据，能够看到后来很多客户端到服务器端连接的标志位只有SYN，没有ACK，说明三次握手没有成功，连接没有建立起来。在socket的通信过程中执行accept()函数时是说明三次握手过程全部完成了，当前三次握手没有完成，于是去accept函数附近查看代码。定位到是因为ET模式下读操作没有放到while循环里面导致的。因为我的epoll使用的是ET模式，当负责监听的文件描述符listenfd触发可读事件时，就证明有新的连接到来，这时我调用了accept进行接收，如果同时有很多新的连接到了，因为没有放到while循环里面，listenfd只会触发一次,只将一个新连接接收进来，大量半连接没有被处理就导致半连接队列满，所以就导致了后面一系列的bug。(应该是全连接队列满了还是半连接队列满了。)

然后通过netstate -tupln | gerp 1316，来查看服务器的端口状态，发现处于Listen状态，再次说明错误应该是发生在三次握手完成之前，

(然后将accept用while循环接收直到返回错误为EAGAIN或者这EWOULDBLOCK为止。)

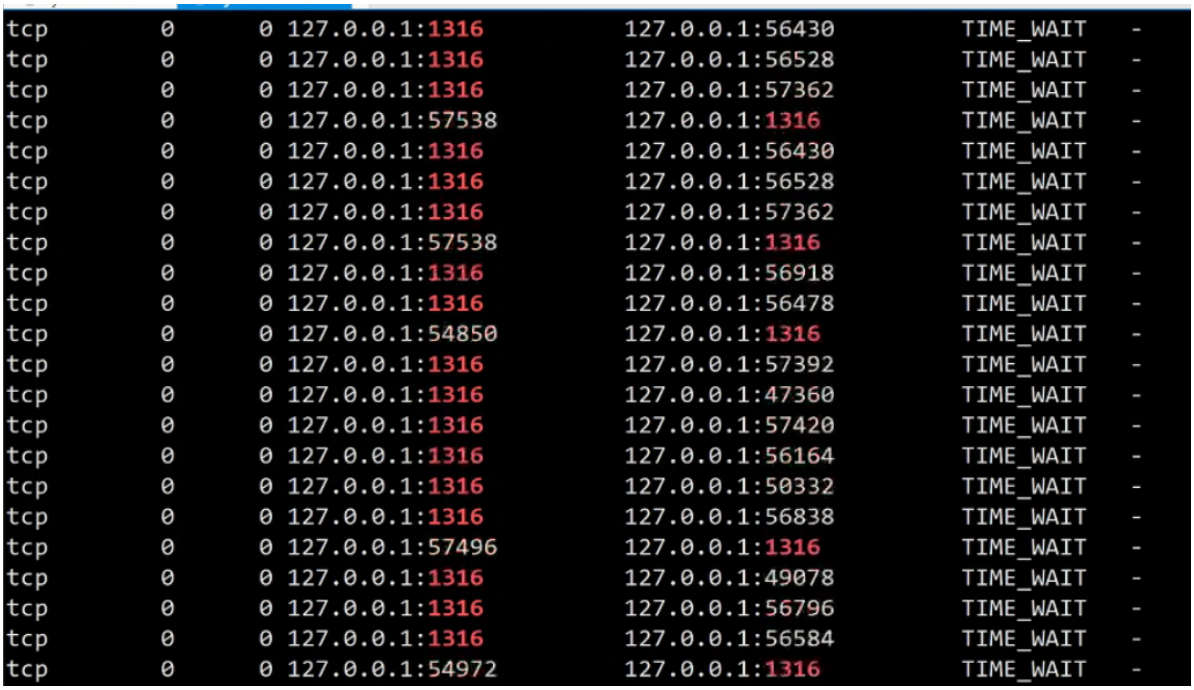
并发量设置为8000时的结果



说Resource temporarily unavailable

通过netstate命令查看端口状态：netstate —anp |grep 1316(服务器端口号)

可以看到很多都是处于time_wait状态的连接



```
tcp        0      0 127.0.0.1:1316          127.0.0.1:56430         TIME_WAIT   -
tcp        0      0 127.0.0.1:1316          127.0.0.1:56528         TIME_WAIT   -
tcp        0      0 127.0.0.1:1316          127.0.0.1:57362         TIME_WAIT   -
tcp        0      0 127.0.0.1:57538         127.0.0.1:1316          TIME_WAIT   -
tcp        0      0 127.0.0.1:1316          127.0.0.1:56430         TIME_WAIT   -
tcp        0      0 127.0.0.1:1316          127.0.0.1:56528         TIME_WAIT   -
tcp        0      0 127.0.0.1:1316          127.0.0.1:57362         TIME_WAIT   -
tcp        0      0 127.0.0.1:57538         127.0.0.1:1316          TIME_WAIT   -
tcp        0      0 127.0.0.1:1316          127.0.0.1:56918         TIME_WAIT   -
tcp        0      0 127.0.0.1:1316          127.0.0.1:56478         TIME_WAIT   -
tcp        0      0 127.0.0.1:54850         127.0.0.1:1316          TIME_WAIT   -
tcp        0      0 127.0.0.1:1316          127.0.0.1:57392         TIME_WAIT   -
tcp        0      0 127.0.0.1:1316          127.0.0.1:47360         TIME_WAIT   -
tcp        0      0 127.0.0.1:1316          127.0.0.1:57420         TIME_WAIT   -
tcp        0      0 127.0.0.1:1316          127.0.0.1:56164         TIME_WAIT   -
tcp        0      0 127.0.0.1:1316          127.0.0.1:50332         TIME_WAIT   -
tcp        0      0 127.0.0.1:1316          127.0.0.1:56838         TIME_WAIT   -
tcp        0      0 127.0.0.1:57496         127.0.0.1:1316          TIME_WAIT   -
tcp        0      0 127.0.0.1:1316          127.0.0.1:49078         TIME_WAIT   -
tcp        0      0 127.0.0.1:1316          127.0.0.1:56796         TIME_WAIT   -
tcp        0      0 127.0.0.1:1316          127.0.0.1:56584         TIME_WAIT   -
tcp        0      0 127.0.0.1:54972         127.0.0.1:1316          TIME_WAIT   -
```

8.EPOLL相关

7.1 IO复用

IO复用：I/O 多路复用技术会用一个系统调用函数来监听我们所有关心的连接，也就说可以在一个监控线程里面监控很多的连接。Linux主要使用select、poll、epoll这三个函数来实现IO多路复用。

7.2 EPOLL原理

原理：主要是使用epoll提供的三个函数，调用epoll_create函数直接在内核区创建一个epoll对象(结构体)，这个epoll对象包含一个红黑树，它存放需要检测的文件描述符(rbr)，还包含一个链表，用来存放那些有事件发生已经就绪的文件描述符。然后调用epoll_ctl()可以注册新的fd到epoll对象中、也可以从epoll对象中删除一个fd、或者修改已经注册在epoll对象中的fd的监听事件。最后调用epoll_wait()用来检测所添加的文件描述符是否有事件到达，内核会

把有事件发生的文件描述符和相应的事件从epoll对象的链表上复制到epoll_wait的形参数组里。

7.3 sel/poll/epo

7.3.1选择epoll原因

1、epoll直接在内核区创建事件表，可减少将事件表从用户态拷贝到内核态的开销。

2、不用每次都遍历事件表来检测就绪事件，就绪的事件会自动存放到epoll_wait传出参数的那个数组里面

3、能够检测的文件描述符没有最大数量限制

4、支持ET和LT两种模式

7.3.2 三者区别

(1)select是先创建一个文件描述符表，然后调用fd_set函数把要检测的文件描述符加入到文件描述符表，让内核帮助我们检查这几个文件描述符是否有数据到达，内核检查完再把这张表返回到用户区，用户遍历这张表找到就绪的文件描述符。

select有四个缺点:

第一是每次调用select都需要把文件描述符表从用户区拷贝到内核区，需要一定的开销。

第二是每次调用select都需要在内核遍历所有传递进来的文件描述符，这个操作也是需要一定开销。(不是1024个，而是要检测的文件描述符最大编号，在这个范围内检测)

第三是select支持的文件描述符数量有限制，最多是1024。

第四是select的文件描述符表不能重复使用，经过内核检查后，表中文件描述符的状态可能会发生变化(某一位置0或置1)，所以每次都需要重置。

(2)而poll与select的不同，第一是poll是通过一个polled结构体向内核传递需要检测的文件描述符，它对文件描述符的数量没有限制。第二是，pollfd中的events成员和revents成员分别表示哪些是需要监听的文件描述符和哪些是有事件发生的文件描述符，故pollfd数组不需要被重置。

(3)而epoll直接在内核区创建事件表(epoll对象)，把要检测的文件描述符放到这个事件表里(epoll对象的红黑树里)，减少了一次将文件描述符表从用户态拷贝到内核态的开销。

(4)select和poll都需要采用轮询的方式来检测就绪事件，时间复杂度是O(n)(自己不断轮询内核检查完返回回来的文件描述符表来找到就绪的文件描述符)，而epoll会把就绪的文件描述符放到内核态的就绪链表里采用回调的方式来检查就绪事件,时间复杂度是O(1)(我们直接将就绪链表拷贝到epoll的形参数组中就可以得到当前就绪的文件描述符。)

epoll对所能检测的文件描述符数量也没有限制,他所支持的文件描述符的数量是最大可以打开文件的数目(最多是65525这么多)。

epoll支持ET和LT两种模式，而poll和select只支持LT模式。

7.4 Select、Epoll应用场景

一般都用select和epoll进行比较，因为select和poll在本质上区别不大

1、select几乎支持所有的平台，相比于epoll，它的跨平台性更好，如果有这方面的要求应选择select。

2、当监测的fd数目小，且各个fd都比较活跃时，使用select(或poll)效率会更高。

3、当监测的fd数目非常大，成千上万那种，并且单位时间内只有一部分的fd处于就绪状态，这个时候使用epoll能够明显提升性能。

原因是select采用轮询的方式检测就绪文件描述符，每次调用都会扫描文件描述符表，再将就绪的文件描述符返回给用户程序。而epoll采用的是回调的方式，内核检测到就绪的文件描述符时，会触发回调函数，回调函数会把就绪的文件描述符对应事件插入到内核的就绪队列，之后再将该就绪队列中的内容拷贝到用户空间。如果在文件描述符数量少，并且活跃的情况下使用epoll，回调函数会被频繁触发，占用系统资源，所以效率就不如select。

7.5 ET和LT

7.5.1 ET/LT区别

LT水平触发模式和ET边缘触发模式是epoll的两种工作模式。

LT模式下，一次事件会被触发多次，比如一个客户端发消息，只要这个客户端的fd(可以认为是客户端的socekt)有数据可读，每次epoll_wait都会返回它的事件，提醒用户程序去处理。

优点：保证了数据的完整输出；
缺点：当数据较大时，需要不断从用户态和内核态切换，消耗了大量的系统资源，影响服务器性能；

ET模式下，一次事件只会触发一次，比如一个客户端发消息，这个客户端的fd有数据可读，应用程序应当立即去处理，如果不处理这个fd，再次调用epoll_wait时，将不再通知这个fd有数据可读。

优点：效率更高，一个事件不会重复的去触发，适合并发量大的场景

缺点：可能会丢失数据，这次事件达到没有处理，下次不会再通知应用程序了

7.5.2 ET下fd为什么设成非阻塞：

因为ET模式下我们必须要用while循环来把客户端文件描述符这次可读的数据全都读出来，否则之后epoll_wait将不会再通知我们去处理这部分数据，正是因为我们的读操作是在while循环中进行的所以一定要将客户端文件描述符设置为非阻塞，如果设置为阻塞，while循环中的readv函数读不到数据就会将线程一直阻塞在这里，不往下进行（while循环卡住），直到再有数据到达，才继续执行，非阻塞下readv会返回一个值，然后继续往下进行。

7.5.3 ET/LT适用场景

ET更适合高并发的场景，ET模式下调用epoll_wait对有可读或可写事件发生的客户端文件描述符只会通知应用程序一次去对其进行处理，效率比较高。

LT下如果一次没有将数据全部处理完，之后调用epoll_wait还会通知应用程序继续读写那个文件描述符的数据，如果系统中有很多你不关心的读写就绪的文件描述符，每次调用epoll_wait，它们都会通知你去处理，降低效率。

7.5.4 epoll底层结构

调用epoll_create，在内核中创建了一个epoll实例，这个实例中包含两部分，一个是用于存放待检测文件描述符的红黑树，另一个是存放有事件发生的就绪文件描述符的双向链表，在调用epoll_wait的时候会把链表里面就绪的文件描述符复制到epoll_wait的形参数组里面，我们通过检测此数组元素就可以找到就绪的文件描述符。

7.6怎么判断客户端断开了连接

项目中检测用于通信的文件描述符对应的epoll事件是**EPOLLRDHUP**时，就说明客户端关闭了连接

7.7 EPOLLONESHOT

当用epoll监听时，如果监听到写/读事件，会从线程池中取一个线程去处理，如果线程处理的不及时，那么epoll就会再唤醒另外一个线程去处理这个事件，造成多个线程处理一个事件的情况。而使用EPOLLONESHOT, 当epoll监听到一个事件发生后并唤醒一个线程后，会将其从epoll监听集删除，也就不会出现多个线程处理一个事件的情况。

7.8 EPOLLRDHUP

EPOLLRDHUP是内核2.6.17后才有的，该事件作用是若对端连接断开时，触发此事件，在底层对对端断开进行处理（之前是在上层通过Recv函数返回值判断）

9.Reactor

1 Reactor/Proactor

Reactor是:（同步IO）

要求主线程（IO处理单元）只负责监听文件描述符上是否有事件发生，有的话就立即将该事件通知工作线程（逻辑单元），将socket可读可写事件放入请求队列，交给工作线程处理。除此之外，主线程不做任何其他实质性的工作。读写数据，接受新的连接，以及处理客户请求均在工作线程中完成。

(结合项目的)

主线程往epoll内核上注册客户端fd读事件，主线程调用epoll_wait等待客户端fd上有数据可读，当fd上有数据可读的时候，主线程把fd可读事件放入工作队列。睡眠在工作队列上的某个工作线程被唤醒，将数据读到用户缓冲区，再处理业务逻辑(解析报文)，读事件处理好以后再往epoll内核上注册fd写请求事件。主线程调用epoll_wait等待写请求事件，当有事件可写的时候，主线程把fd可写

事件放入工作队列。睡眠在请求队列上的工作线程被唤醒，执行写事件。

Proactor是:（异步IO）

主线程调用aio_read函数向内核注册fd上的读完成事件，并告诉内核用户读缓冲区的位置，以及读完成后如何通知应用程序，然后主线程继续处理其他逻辑，当fd上的数据被读入用户缓冲区后，通过信号告知应用程序数据已经可以使用。应用程序预先定义好的信号处理函数选择一个工作线程来处理客户请求。工作线程处理完客户请求之后调用aio_write函数向内核注册fd写完成事件，并告诉内核写缓冲区的位置，以及写完成时如何通知应用程序。主线程处理其他逻辑。当用户缓存区的数据被写入fd之后内核向应用程序发送一个信号，以通知应用程序数据已经发送完毕。应用程序预先定义的数据处理函数就会完成工作。

2 两者区别：

Reactor中需要应用程序自己将数据从内核缓冲区读入用户缓冲区，或将数据从用户缓冲区写入内核缓冲区，而Proactor模式中，应用程序不需要用户再自己接收数据，直接使用就可以了，操作系统会将数据从内核拷贝到用户区 。简单说就是Reactor模式下向应用程序通知的是I/O就绪事件，Proactor模式下向应用程序通知的是I/O完成事件。

3 Reactor的好处

相较于多线程并发模型，IO多路复用的并发模型不用为每一个客户开启一个线程，所以没有线程同步问题和线程切换开销。

Linux系统对异步IO的支持不是很完善，大部分网络库的实现都是基于同步IO的Reactor模型，所以我这里也使用的网络库。

10.五种IO模型

分别为阻塞IO、非阻塞IO、信号驱动IO、IO复用、异步IO

阻塞IO：调用者调用了某个函数，必须等到这个函数返回了以后才能继续向下执行，在这期间不停的去检查这个函数是否返回。

非阻塞IO：调用者调用了某个函数，每隔一段时间就去检查IO事件是否就绪，没有就绪就去做其他的事情。

信号驱动IO：安装一个信号处理函数，进程继续运行并不阻塞，当IO事件就绪，进程收到信号，然后处理IO事件。

IO复用：Linux使用select、poll、epoll函数来实现IO多路复用，他们可以同时对多个IO操作进行检测。直到有数据可读或可写时，才真正调用IO操作函数。

异步IO：由操作系统内核来完成IO操作，同时内核会将数据从内核缓冲区拷贝到用户缓冲区，然后通知应用程序，用户可以直接使用这些数据。

11.同步异步

1.线程同步/异步

线程同步是指多个线程之间执行具有先后顺序，或者因果关系，当有一个线程对某块内存进行操作时，其他线程必须等该线程执行完了，才能对这块内存进行操作，其他线程一直处于等待状态。

线程异步指多个线程之间的执行，没有先后关系，不需要互相等待，各做各的。

2.IO同步/异步

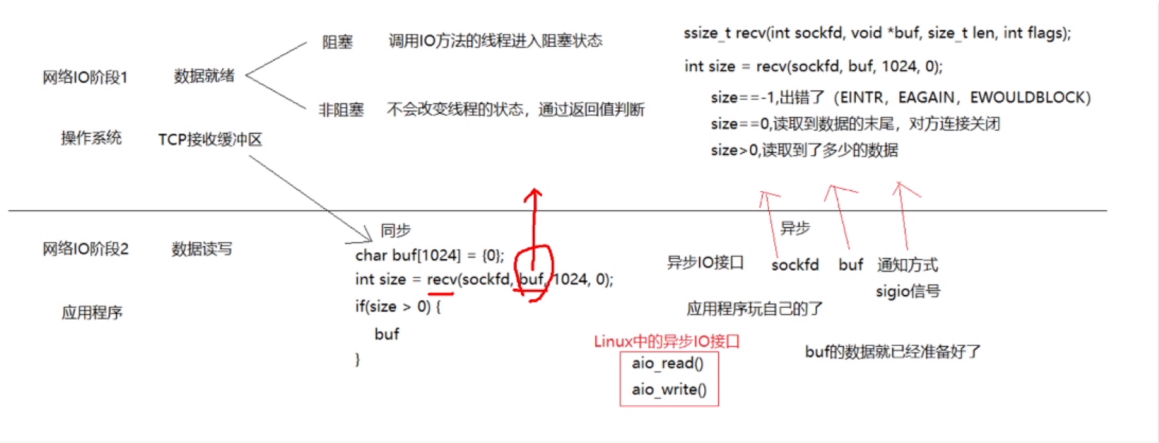
同步IO模型要求用户代码自行将数据从内核区读入到用户区，或者从用户区写入到内核区，而异步IO是由内核完成IO操作，内核帮你完成数据在用户区和内核区之间的移动，简单说就是同步IO向应用程序通知的是IO就绪事件，异步IO向应用程序通知的是IO完成事件

(以我们项目为例，我们用的同步IO来实现Reactor模型，所以主线程检测到客户端有数据到达时，它把任务加入到工作队列，某个空闲的子线程从队列中取这个任务，它需要先自己读数据，然后再去做业务逻辑)。

3.IO阻塞/非阻塞

IO阻塞与非阻塞是相对于系统IO操作的就绪状态来说的，以recv()函数为例，阻塞就是指若没有数据就绪/到达，则程序会一直阻塞在这个函数上直到有数据到达；非阻塞就是通过设置flag为O_NONBLOCK，即`fcntl(fd, F_SETFL, fcntl(fd, F_GETFD,`

0) | O_NONBLOCK);当运行到recv函数时不会再阻塞等待数据到达，而是立即返回值，通过判断返回值来判断数据就绪状态



12.项目优化方面

1、可以考虑把现在线程池中线程执行IO操作的任务分离出来，而在主线程中再创建一个线程池专门用于做IO操作，这样做可以减少工作线程的压力，并发量应该会得到提高。

2、锁方面的优化

(1)可以考虑用读写锁来替换互斥锁，因为我们当前业务逻辑是分为读任务(解析请求报文)和写任务(返回响应报文)，读写锁的粒度要比互斥锁更低，这样线程的执行速度会比较快。

(2)可以考虑无锁编程，我们当前使用的是互斥锁，锁这部分引入的代码不仅不处理任何业务逻辑，而且在使用锁的时候还需要进行线程的切换，造成一定的系统资源开销。

CAS(compare and swap)

感觉通过CAS这种原子操作，在不使用锁的情况下，可以实现在用户区来保证多线程共享数据的安全。

CAS(对比并交换）原理：CAS是一个原子指令，用于在多线程环境中实现同步。 它包括三个操作数：内存位置(V)、预期原值(A)、新值(B)。如果内存位置的值与预期原值相匹配，那么处理器会自动将该位置值更新为新值(说明没被其他线程修改，现在可改)，否则，不做任何操作。这个过程是作为单个原子操作完成的。原子性保证了这个新值是最新的，这个新值在被读取至操作完成过程中不会被其它线程修改。

(3)还有一种取消锁的方式就是，我每个子线程都弄一个任务队列，主线程通过负载均衡的方式给任务分配到子线程的任务队列里面，这样就没有多个线程抢夺资源的问题，也就不用锁了。

第一种 轮询（默认）每个请求按时间顺序逐一分配到不同的后端服务器，如果后端服务器 down 掉，能自动剔除。 **第二种 weight，根据upstream机制实现**weight 代表权重默认为 1,权重越高被分配的客户端越多 **第三种 ip_hash** 每个请求按访问 ip 的 hash 结果分配，这样每个访客固定访问一个后端服务器

15.数学建模

讲一下你们数学建模竞赛的情况以及你在其中担任的角色？做了哪些工作?

我们当时在数学建模竞赛中选择的是D题。我在这其中主要担任的是对讨论得到的解题思路使用Matlab进行编程实现并输出最终结果的工作。D题给了大量的关于乳腺癌化合物生物活性及分子描述符的数据，问题是首先要我们对这些数据进行筛选，然后根据筛选出的数据（分子描述符）建立预测模型以及分类预测模型，我们具体

是先通过皮尔逊相关性分析筛选出数据后，通过BP神经网络建立了预测模型，并且通过Relief进行特征分析，然后再通过随机森林算法得到了分类预测模型（第三问），进而解决了竞赛的一系列问题。